CLARAty
Mechanism Model Software Design Document

Revision: 0.9d (*Draft*)
Date: March 28, 2005

Prepared By:
Issa A.D. Nesnas
Won S. Kim
Hari Das Nayar
Antonio Diaz-Calderon

Document Custodian:
Issa A.D. Nesnas

Contributors
Anne Wright (Ames Research Center)
Raymond Cipra (Purdue University)
Max Bajracharya (JPL)
Daniel Clouse (JPL)

Paper copies of this document may not be current and should not be relied on for official purposes. The current version is on the CLARAty website at http://claraty.jpl.nasa.gov under Software/Packages/Mobility and Manipulation

Jet Propulsion Laboratory
4800 Oak Grove Drive
Pasadena, CA 91109-8099

This Page Intentionally Left Blank

CLARAty Mechanism Model Requirements and Design Document

Signature Sheet

Approval

Mr. Clay Kunz – CLARAty Center Lead, NASA Ames Research Center Date

Dr. Stergios Roumeliotis – CLARAty Center Lead, University of Minnesota Date

Dr. Reid Simmons – CLARAty Center Lead, Carnegie Mellon Date

Dr. Antonio Diaz Calderon – Jet Propulsion Laboratory Date

Dr. Hari Das Nayar – OphirTech Date

Dr. Won S. Kim – Activity Lead, Jet Propulsion Laboratory Date

Dr. Issa A.D. Nesnas – CLARAty Task Manager, Jet Propulsion Laboratory Date

Revision: 0.9d Draft
Date: March 28, 2005

Table of Contents

Revision History

Revision	Date	Description	Author
0.1	02/02/04	Initial document to capture meeting notes	W. S. Kim
0.2	02/26/04	Added provisions for legged mechanisms; collision modeling; and parameter files	W.S. Kim
0.3	03/01/04	Revised collision models and added kinematic model methods	W.S. Kim
0.4	04/26/04	Restructured the document into functional and interface reqs; added figures	W.S. Kim
0.5	05/14/04	Added definitions for body and joint; added coordinate frames; usage models; and captured unresolved items for further discussion	I.A. Nesnas
0.6	06/09/04	Restructured document; made major additions to all sections; added modeling info from A. Diaz and design diagrams from H. Nayar and W. S. Kim	I.A. Nesnas
0.7	06/23/04	Added introduction, body tree section, closed chains, body tree, and bounding shape diagrams; updated usage model section	I.A. Nesnas
0.8	06/25/04	Added body joint relationship diagram; cleaned up and reorganized some sections	I.A. Nesnas
0.9	10/08/04	Added constraint management section from R. Cipra and reviewed several sections	H. Nayar
0.9a	12/30/04	Prepared document for final review; added cover pages, redid figures; added table of contents, added missing figures from version 0.9; cleaned up styles and updates sections	I.A. Nesnas
0.9b	3/1/05	Updated mechanism model class hierarchy and mechanism model interface figures	I.A. Nesnas
0.9c	3/1/05	Added Appendix A; reviewed DTD; and reviewed document	I.A. Nesnas
0.9d	3/25/05	Incorporated changes for co-authors: Won, Hari, Antonio. Added comparison section from Hari	I.A. Nesnas

1. Introduction

What is proposed?

A unified approach for modeling mechanical properties of a robotic system for use by the CLARAty[1] on-board software. The implementation of these requirements will provide CLARAty with a more generic infrastructure for mechanism modeling. The modeling software covers mobility mechanisms, robotic arms, rover masts, mechanical legs, and so on. The modeling software will provide the necessary information for real-time computation of kinematics, dynamics, and collision prediction.

Why is it proposed?

A unified modeling approach has the following advantages:

- Provides centralized storage for managing model information. This includes creation, deletion, update, extension and reconfiguration of the mechanical models.
- Ensures consistency of the model information for use by multiple algorithms. This will simplify the integration of algorithms into the software architecture.
- Reduces duplication in model representation between rover mobility and manipulation software.
- Enables the development of generic algorithms for forward, inverse, and differential kinematics. In the absence of specialized versions, the generic algorithms provide out-of-the-box functionality.
- Supports specific implementations to override generic algorithms whenever appropriate for optimal performance.
- Enables the verification of specialized kinematics algorithms against their generic counterparts.

What is contained in this document?

This document contains software requirements for developing a unified mechanical model representation. It also contains requirements for algorithms and describes the interaction of the models with the rest of the on-board robotic software.

This document is divided into two parts. Part I covers the mechanical model requirements and design. It is divided into several sections covering general requirements, mechanical models and their components, relationships between these components, and user input for generating the models. Part II covers the integration of these models with the rest of the on-board control software. The appendices include sample model files and various parameter representations.

This document assumes familiarity with robotic terminology and basic knowledge of software development and object-oriented concepts. In this document, the term “mechanism” refers to any mechanical system and does not imply a closed loop mechanical chain.

[1] CLARAty: Coupled Layer Architecture for Robotic Autonomy

Related Work

The Mechanism Model package has similarities to and differences from other kinematics modeling software packages. A review and comparison of Mechanical Model with three other packages: (1) DARTS/Dshell, (2) Open Robot Control Software Kinematics (ORCOS) package and (3) Operational Software Components for Advanced Robotics (OSCAR) is given below.

DARTS/Dshell

Developer:

Abhi Jain and DARTS/Dshell group (JPL)

Summary:

DARTS/Dshell is a high-fidelity dynamics simulator that models the motion of flexible multi-body systems under internal and external interactions. It has been used to model robotic systems and spacecraft. Applications include hardware-in-the-loop testing and off-line simulations.

The development of DARTS/Dshell began in the early 1990s. The current version is implemented in C++ and runs under UNIX/Linux environments. Real-time versions for hardware-in-the-loop testing have been ported to VxWorks. The use of DARTS/Dshell as an application is actively supported at JPL.

Comparison:

DARTS/Dshell and CLARAty share many similar mathematical utilities like vector, matrix, quaternion and transform classes. The modeling approach and software architecture in Mechanism Model is based on the DARTS/Dshell approach. The primary differences between Mechanism Model in CLARAty and DARTS/Dshell are:

- The modeling in DARTS/Dshell is geared for high fidelity simulations while the Mechanism Modeling of CLARAty is geared for real-time high-frequency control and planning. The latter also supports overriding of generalized solutions with specialized ones.
- DARTS/Dshell uses much more detailed models (body flexibility, actuator and transmission modeling, etc.) which are needed for high-fidelity simulations.
- Simulation software and control software solve complementary problems (e.g. simulation software solves the forward dynamics while controls software solves the inverse dynamics).
- DARTS/Dshell makes extensive use of recursive algorithms while Mechanism Model will use corresponding iterative algorithms, which are more amenable for on-board flight implementations. Mechanism Model will allow limited use of recursive approaches in the future.

Open Robot Control Software (OROCOS) Kinematics Package

Developer:

Herman Bruyninckx (Katholieke Universiteit – KU, Leuven, Belgium)
Anthony Mallet (Laboratory for Analysis and Architecture of Systems - CNRS/LAAS, France)
Henrik Christensen (Kungl Tekniska Högskolan - KTH, Sweden)
Other collaborators

Summary:

The original goal of the OROCOS effort was to develop open source software for robotics applications. It has since branched into two separate developments:

- Open Real-time Control Services for real-time control applications
- Open Robot Control Software provides class libraries and a framework for robot applications.

OROCOS's objectives are similar to CLARAty's. The Kinematics Package in OROCOS has similarities to Mechanism Model. However, the Kinematics Package is at the design concept stage and no software has been developed. It is intended to address more general mechanical systems and not be restricted to tree-topology systems. While the objectives of the Kinematics Package have been documented, the approach to be used for its implementation has yet to be clearly defined.

The development of OROCOS began in 2001 as a collaboration between KU Lueven, Belgium, CNRS/LAAS, France and KTH, Sweden. It was funded by the European Union between 2001 and 2003. The work continues as an open source project primarily lead by Herman Bruyninckx. The OROCOS development environment uses C++ as its programming language and Linux and RTAI (a realtime extension to Linux) as its operating system. While OROCOS can be more generally applied, it has had a focus on industrial robotics. The future of the Kinematics Package in OROCOS is unclear because of the open source nature of the development and lack of any funding.

Comparison:

The OROCOS Kinematics package currently exists as a set of objectives without a detail description of its approach or implementation.

Operational Software Components for Advanced Robots (OSCAR)

Developer:

Del Tesar's Robotics Research Group, University of Texas

Summary:

OSCAR provides utilities in the form of libraries for performing computations needed in analysis, control or simulation of manipulators. In addition to math utilities, it contains algorithms for performing generic forward and inverse kinematics, motion planning and dynamics. OSCAR offers many alternative options in its operations. For example, for motion planning, trajectories can be generated using trapezoidal, spline or motion blending algorithms. OSCAR currently appears to allow only models of serial-chain manipulators. OSCAR's primary application is robotics education.

OSCAR is a set of C++ libraries built in the Windows environment using Visual C++. The libraries are used by linking them to an application that is also being developed in Visual C++ under Windows. It was originally developed in the mid-1990s and appears to be actively supported.

Comparison:

OSCAR provides generic software utilities for robot arms (serial-chain manipulators) while Mechanical Model models more general kinematics systems. OSCAR is used exclusively in PC Windows environments and interfaces to application software that is developed in Visual C++.

Part I: Mechanism Model

2. General Requirements

2.1. The software shall separate mechanism models from mechanism control. This allows client software to use and test the kinematics and dynamics of the mechanisms independent of the hardware.

2.2. The software shall support the following computations for the mechanism model:

- 2.2.1. Kinematics computations:
  - 2.2.1.1. Forward, inverse, and differential kinematics
- 2.2.2. Quasi-static computations of forces and torques considering:
  - 2.2.2.1. Joint flexibility (stiffness)
  - 2.2.2.2. Gravity force and other applied forces
  - 2.2.2.3. Gravity deflection
  - 2.2.2.4. Environmental contact constraints (position, force, torque, and stiffness)
- 2.2.3. Resolution of multiple simultaneous kinematics constraints
- 2.2.4. Collision detection
- 2.2.5. Not full dynamics computations (inertial forces). The software shall not support dynamic computations in the initial implementation. However, models shall support future extensions for dynamics computations.

2.3. The software shall allow over-riding of generic algorithms with specialized algorithms for specific kinematics systems.

3. Mechanism Types

3.1. The mechanism model software shall handle multi-body mechanisms which include:

3.1.1. Serial manipulators:
Multi-degree of freedom robotic arms, masts, and legs (e.g., a 5-dof arm with a turret gripper carrying multiple instruments)

3.1.2. Simple closed-chain mechanisms:
Four- and six-bar planar mechanisms

3.1.3. Wheeled locomotors:
Multi-wheeled mechanisms with different drive and steering configurations. This includes fully-steerable, partially-steerable, and skid-steerable mechanisms (e.g. Rocky 8's six-wheel drive six-wheel steering rocker bogie mechanism, Rocky 7's all-wheel drive front wheel steering mechanism, SRR's[2] four-wheel rocker mechanism, and ATRV's[3] non-steerable mechanism)

3.1.4. Legged locomotors:
Multiple legs attached to a body (e.g. LEMUR[4] and Athlete robots). Legged locomotors that require dynamics computation will currently not be supported (e.g. two-legged humanoid robots)

[2] SRR: Sample Return Rover built by JPL
[3] ATRV: All Terrain Response Vehicle from IRobot
[4] LEMUR: Limbed Excursion Mobile Utility Robot built by JPL

3.1.5. Composite mechanisms:
Any combination of the above types (e.g. a rover with a robotic arm; a mobile robot with both wheels and legs)

3.2. The mechanism model software will not directly handle parallel and hybrid kinematic structures. However, it shall be possible to model a simple parallel structure as a tree topology by breaking the closed chain and solving for the closed chain using constraints.

3.3. To verify the viability and fidelity of the mechanism model software package, the following models shall be developed and tested with the new approach:
- 3.3.1. Rocky8's two degree-of-freedom (DOF) fixed mast
- 3.3.2. MER's 5-DOF robotic arm
- 3.3.3. Moonrise's 4-DOF robotic arm for digging operations
- 3.3.4. FIDO's rocker-bogie fully steerable wheeled locomotor
- 3.3.5. LEMUR or Athlete's legged mechanism
- 3.3.6. Four-bar planar mechanism

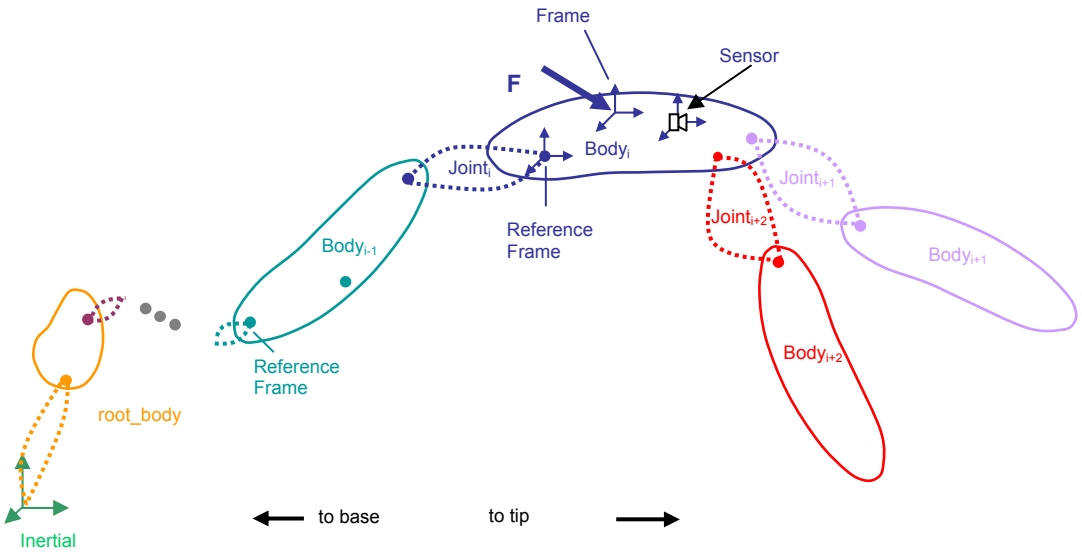


Figure 1: Mechanism Model: bodies and joints

4. Mechanism Model Design

4.1. The software shall capture the mechanism and its associated coordinate frame transformations in a tree topology as shown in Figure 1. We shall represent mechanisms using open-loop chains. Using a tree topology considerably simplifies the software infrastructure and enables both flexible and efficient processing.

4.2. The `Mechanism_Model` shall be the top-level software object. The `Mechanism_Model` class shall provide the interface to the mechanism model database for creating, deleting, modifying, querying, and performing kinematic analysis. We shall use a generic `Tree` class in the `Mechanism_Model` to represent the mechanism topology.

4.3. The mechanism tree shall consist of a number of **bodies** connected to one another via **joints**. A joint connects a body to its parent body. The body will be denoted by `ME_Body` (short for mechanical element body). The joint will be denoted by `ME_Joint` (short for

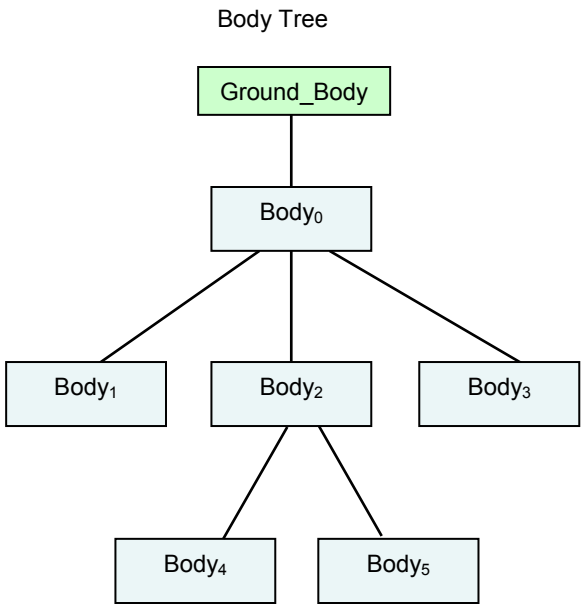
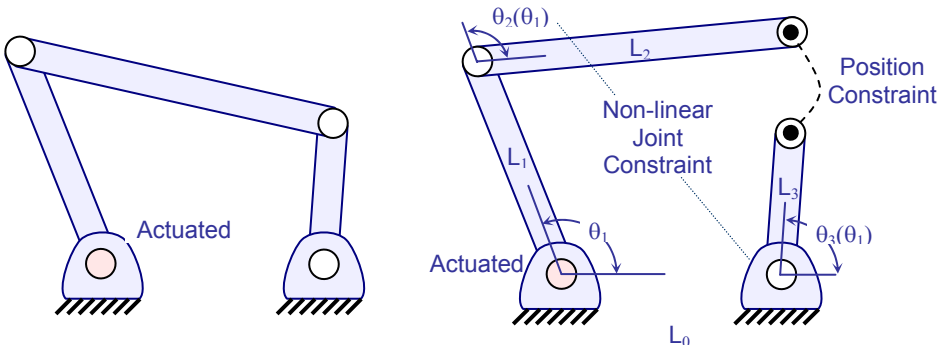


Figure 2: Mechanism Model Body Tree



(a) Four-bar

(b) Four-bar mechanism modeled with either (i) position constraint, or (ii) with non-linear joint constraints

Figure 3: Handling closed loop chains

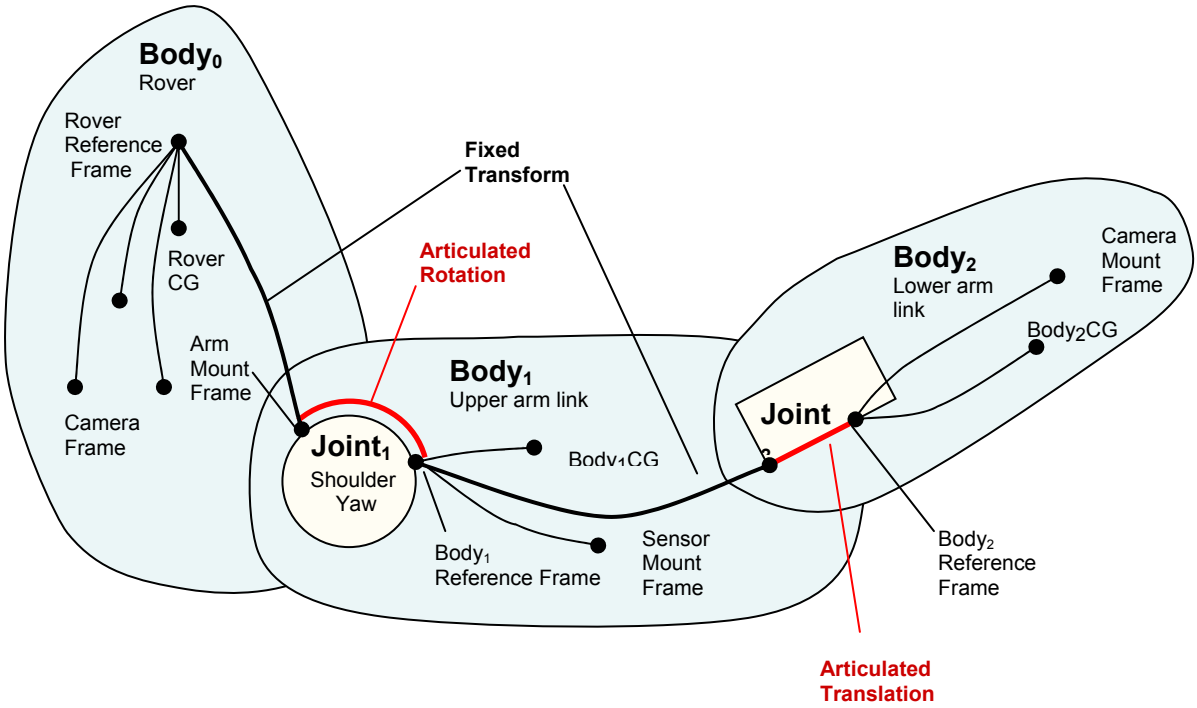


Figure 4: Handling fixed and articulated coordinate frame transformations

mechanical element joint) (Figure 2).

4.4. Closed loop chains such as a four-bar mechanisms, for example, shall be handled as an open loop chain with either a position constraint between two of its links or non-linear joint constraints on the non-actuated joints (see Figure 3).

4.5. The mechanism model tree shall only store mechanical model information. It shall **not** store any joint or state values. Mechanical model information includes fixed (non-articulated) transformations[5] and joint constants that do not depend on

[5] Relationships between coordinate frames that do not move with respect to the body

articulation values (Figure 4). All articulation and state information shall be passed to the mechanism model. Keeping the fixed transformations separated from the articulated transformations (due to relative motion between bodies) allows us to make the tree stateless.

4.6. The mechanism model application program interface (API) shall support inputs from a vector of articulation values (e.g. a vector of joint angles/prismatic values) to compute the articulated transformations.

4.7. The mechanism model tree shall support multiple clients simultaneously (i.e. the mechanism model shall be multi-thread safe and re-entrant).

4.8. Position, velocity, and acceleration information relative to an inertial frame shall not be stored in the mechanism model. If such information needs to be stored, it will be cached in the algorithms that require and compute this information. This is important because it will enable various states to be updated at different rates and enable the use of parts of the tree at a time. It will also allow algorithms to use the mechanism model tree to predict future states for any given input state. The trade that is made here is the cost of re-computing derived states vs. making copies of mechanism model for each client application and keeping all their internal state up to date.

4.9. There shall be a single inertial frame denoted by `Ground_Body` in a given deployment of systems. `Ground_Body` is the root of the tree (for a single or even multiple robots).

4.10. The `Ground_Body` shall have no joint associated with it because it is at the root of the tree and does not need a parent. It can have multiple transformations denoting mounting locations of interest in the site. Multiple mechanisms could reference the same `Ground_Body`. The location of a mechanism relative to the `Ground_Body` is defined in the joint that connects that body to the `Ground_Body`.

4.11. The `Ground_Body` will typically not be part of a particular mechanism model description (i.e. the model input file (see Section 6)). However it can be referenced by the model description. Usually the relationship or location between a `Mechanism_Model` and the `Ground_Body` is defined during the model instantiation. This is important for creating composite mechanisms from identical components. For example, if you have a six-legged robot with identical legs, you only need a model for one of the legs and you can then create the robot by changing the mounting point for each leg.

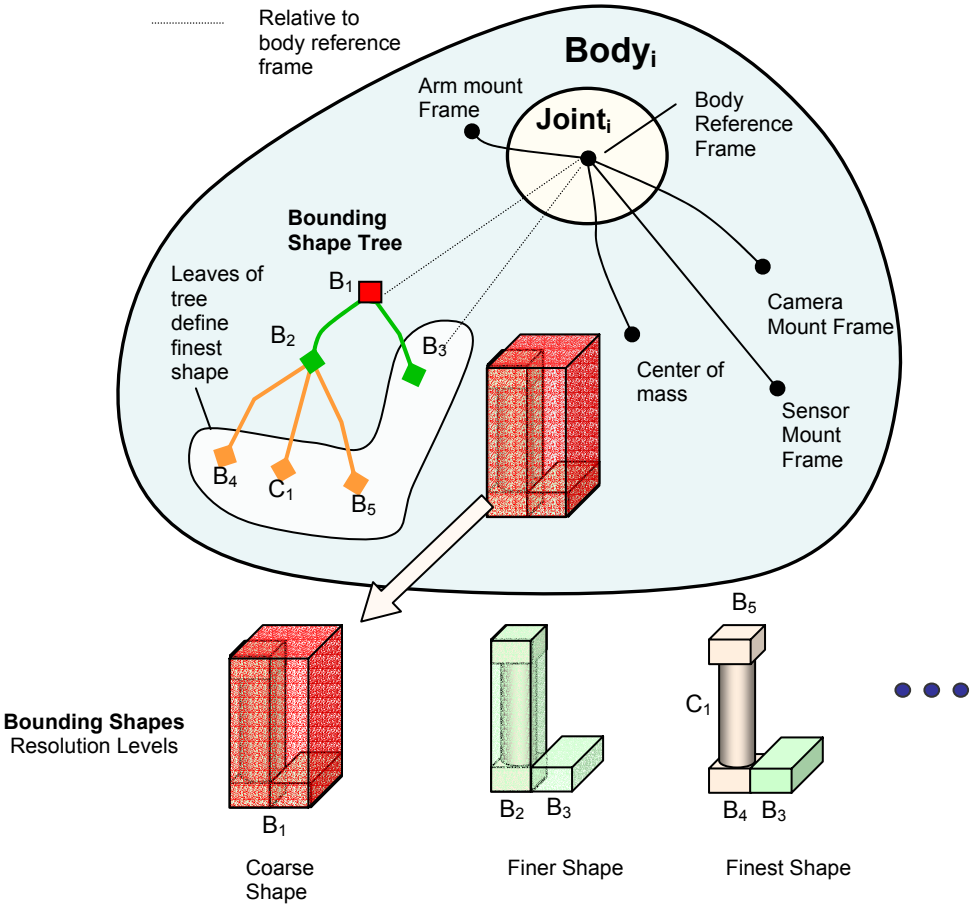
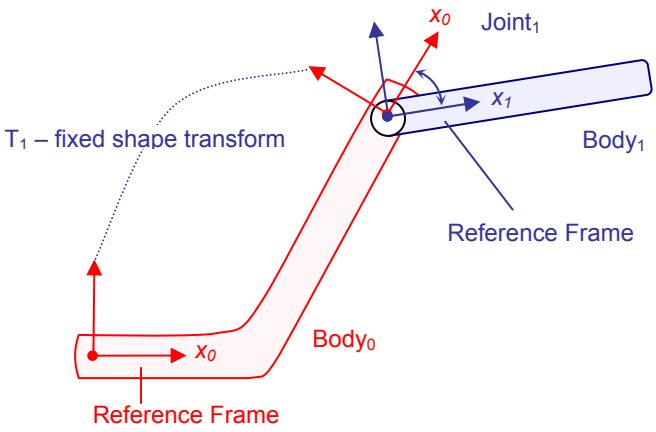


Figure 5: The components of a mechanical element body: `ME_Body`



Body$_1$ object contains joint$_1$
Body$_1$ object contains fixed shape transform defining joint$_1$ zero location
Body$_1$ reference frame z-axis is aligned with joint$_1$ rotation axis

Figure 6: Defining relationship between body and joint in a mechanism model

4.12. There shall be a single tree of bodies that will capture the mechanism model and the bounding shape information needed for collision detection. The mechanism model API shall support the copying of bounding shape objects (e.g. trees) for collision detection algorithms. Copies of bounding shapes can store transforms relative to an internal frame (Figure 5).

4.13. An `ME_Body` shall have the following characteristics:

- 4.13.1. A single parent body. This simplifies the internal representation without any loss of generality.
- 4.13.2. A single joint that connects a body to its parent (i.e. open loop chain). The only exception is the `Ground_Body` (root of the tree), which has no parent and hence no joint.
- 4.13.3. A fixed shape transformation that defines the joint location relative to its parent's reference frame. The relationships between bodies, joints and their coordinate frames are illustrated in Figure 6. The joint location frame shall have its z-axis aligned with the actuation axis. For revolute joints, the angle of rotation is about the +z-axis following the right-hand rule. For prismatic joints, the translation is along the z-axis. The x-axis of the joint location is user defined. In most cases it is aligned with the body's length.
- 4.13.4. The body reference frame is attached to the body and is coincident and aligned with the fixed shape transform (joint) frame of the parent when the joint is in zero position. The joint articulation transform represents the offset from the zero configuration.
- 4.13.5. Any number of fixed (non-articulated) transformations defined relative to the body reference frame (Figure 5).
- 4.13.6. A textual representation of the body name.

4.14. Bodies are assumed to be rigid

- 4.14.1. Software shall be extendable to support flexible body models.

4.15. An `ME_Body` may have the following optional characteristics:

- 4.15.1. Representation of bounding shape information for use by collision detection algorithms:
  - 4.15.1.1. A body contains a bounding shape tree (see Figure 5) that describes containment relationships among the geometric objects of a single body.
  - 4.15.1.2. Algorithms shall specify and have access to different resolutions of the bounding shape information. The finest bounding shape resolution shall be at the leaves of the tree.
  - 4.15.1.3. Bounding shapes in each body shall have their position transformations relative to the body reference frame.
  - 4.15.1.4. The corresponding `ME_Body` shall manage (update, delete) bounding shapes.
  - 4.15.1.5. We shall represent bounding shapes either as 2D or 3D shapes (e.g. we represent terrain surfaces and walls by 2D open meshes and manipulator links by 3D shapes such as cylinders, boxes, spheres, and/or convex hulls). The bounding shape API shall support both 2D and 3D representations[6].
- 4.15.2. Representation of geometric information for graphics display (future implementation).

4.16. An `ME_Joint` shall have the following characteristics:

[6] Bounding shape support for oct trees needs further investigation

4.16.1. A joint must not store its articulation value or any joint state information (such as joint mode which enumerates as: servoing, free, locked, etc.) – i.e. a joint must be stateless. State information refers to state that will change over time. Joint state is stored outside the tree. The articulation value for a revolute joint is defined by the rotation about the +z-axis relative to the fixed transform.

4.16.2. A function that accepts joint state as input and returns the relative transformation between the current body reference frame and the parent's reference frame.

4.16.3. A field defining the joint type: active (actuated) or passive (non-actuated); revolute, prismatic, ball, or planar (not to be confused with joint state).

4.16.4. Optional fields for specifying joint limits: min and max values

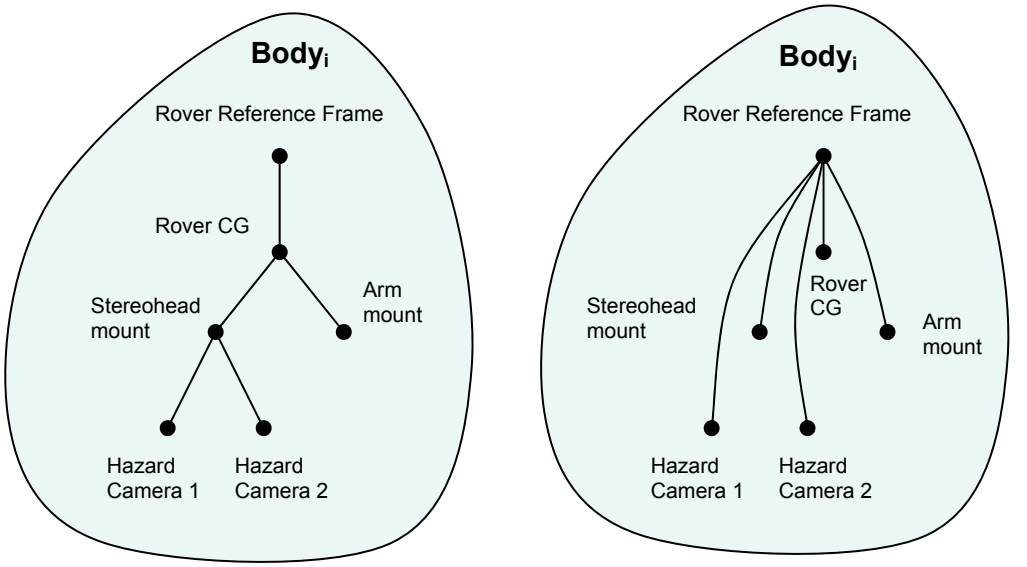
4.17. An `ME_Joint` may have the following optional characteristics:

4.17.1. Fields for defining joint constraints – i.e. constraints that couple a joint to another (e.g. joint2 = a*joint1 + b) represented by the `Joint_Constraint` software object.

4.17.1.1. Joints shall have built-in support for linear constraints.

4.17.1.2. Joints shall support extensions for non-linear constraints.

4.17.2. Fields for specifying joint stiffness with linear parameters [kx ky kz tx ty tz].



(a) Fixed coordinate frame transformation can be inputted from model file (XML) relative to one another

(b) All fixed frames computed and internally stored relative to body reference

Figure 7: Options for coordinate frame transformation input vs. internal storage representation

5. Coordinate Frames

5.1. A coordinate frame transformation represents the relative position and orientation of one coordinate frame relative to another. A coordinate frame transformation will be denoted by `Transform`.

5.2. A `Transform` shall have an API similar to the one currently implemented by `HTrans` in CLARAty (homogeneous transform). The `HTrans` internally represents a coordinate transformation as a 3x3 rotation matrix and a 1x3 translation vector. However, its API supports matrix operations as if it were a 4 x 4 matrix without the cost of additional computation.

The rotation portion of `Transform` shall be implemented using the `Quaternion` rotation class. The translation portion of the `Transform` class shall be implemented using the `Point` (3D point) class.

5.3. A `Frame` object shall be used to denote the physical location of a coordinate frame relative to its body reference frame. The `Frame` object shall contain a `Transform` object to specify its position and orientation relative to its body reference `Frame`, a reference to its `Body` object, and a textual string to specify its name.

5.4. While body-referenced coordinate frames may be relative to intermediate coordinate frames in the data input representation, the software shall internally compute and store all these coordinate frames relative to the body reference (Figure 7).

6. Model Data Input

6.1. Mechanism model parameters shall be specified in an eXtensible Markup Language (XML) input file. Appendix A contains an example XML file and its corresponding Document Type Definition (DTD).

6.2. CLARAty shall use SI units of meters for lengths, radians for angles, and kilogram for mass in the input file. All internal values shall be stored in SI units. The implementation shall easily be extended to support more general input units.

6.3. Each mechanism or appendage shall have a separate XML file. The application program shall read multiple files and construct the system model. The complete mechanism model may be assembled by reading in multiple XML files (e.g. arm model, mast model, and mobility model are stored in separate files).

To simplify our initial implementation, we will require a certain order for reading input files to attach appendages to proper mount points. For example, a rover body model file will be read before an arm model file in order to attach the arm to the rover body.

6.4. The mechanism model file shall define necessary mount points by name to attach appendages defined in separate model files.

6.5. Mechanism model shall, if necessary, override mounting information defined by the model file and specify different mounting points during instantiation.

6.6. Mounting information for the base of appendages relative to the inertial frame should not belong to the appendage (arm) model files. This information is specified when attaching the appendage to the mechanism model. A given appendage may be mounted differently on various systems. For example: a K9 arm may have a different mounting location in the lab than on the K9 rover.

6.7. XML input file format shall support fixed coordinate frames relative to other fixed frames in a rigid body (Figure 7)[7]. However, the software shall internally compute and store all these coordinate frames relative to the body reference frame (see requirement 5.1)

6.8. Software shall support serialization (marshalling / de-marshalling) of mechanism model information for transmission/storage over media. Software shall use CLARAty's mechanism's for marshalling/de-marshalling using Flexible Data Marshalling (FDM) package in the `data_io` module for reading and writing objects and data from and to a variety of formats. The following formats shall be supported:

- 6.8.1. Text tagged XML and Parse Block.
- 6.8.2. Binary tagged ACE CDR and untagged file IO.

6.9. Input parameter file shall support the required kinematic parameters.

6.10. Input parameter file shall support the optional parameters of:

- 6.10.1. Center of mass
- 6.10.2. Inertia matrix
- 6.10.3. Bounding shape

6.11. Input parameter file shall support different representations to specify the model (see Appendix B):

- 6.11.1. Homogenous Transform (HT)
- 6.11.2. Zero Position (ZP)
- 6.11.3. Denavit-Hartenberg-Craig (DHC) (per J.J. Craig) for systems composed of one DOF revolute joints

[7] Similar to what the Frame Tree implementation currently supports.

6.11.4. Denavit-Hartenberg-Paul (DHP) (per R. Paul) only for serial manipulators composed of one DOF revolute joints. Note that DHP does not support branches.

6.12. Model parameters are converted to an internal representation as follows:
- 6.12.1. For single degree-of-freedom joints, the z-axis shall be aligned with the articulation axis.
- 6.12.2. For multiple degrees-of-freedom joints, the z-axis shall be aligned with at least one joint axis.
- 6.12.3. The body reference frame shall be located at the center of rotation of the revolute joints.

6.13. Software shall support saving or streaming the model information to a file in a format that matches the input representations. General conversion between different model representations might not be possible for some representations.

7. Kinematic Algorithms

7.1. **Forward Kinematics:** The `Mechanism_Model` object shall implement a function to compute the transformation between any two `Frame`s in the mechanism model for a given set of corresponding articulation values.

7.2. **Differential Forward Kinematics:** The `Mechanism_Model` object shall implement a function to compute the linear and angular velocities of any `Frame` in the kinematic system with respect to another `Frame` for a given set of articulation values and velocities.

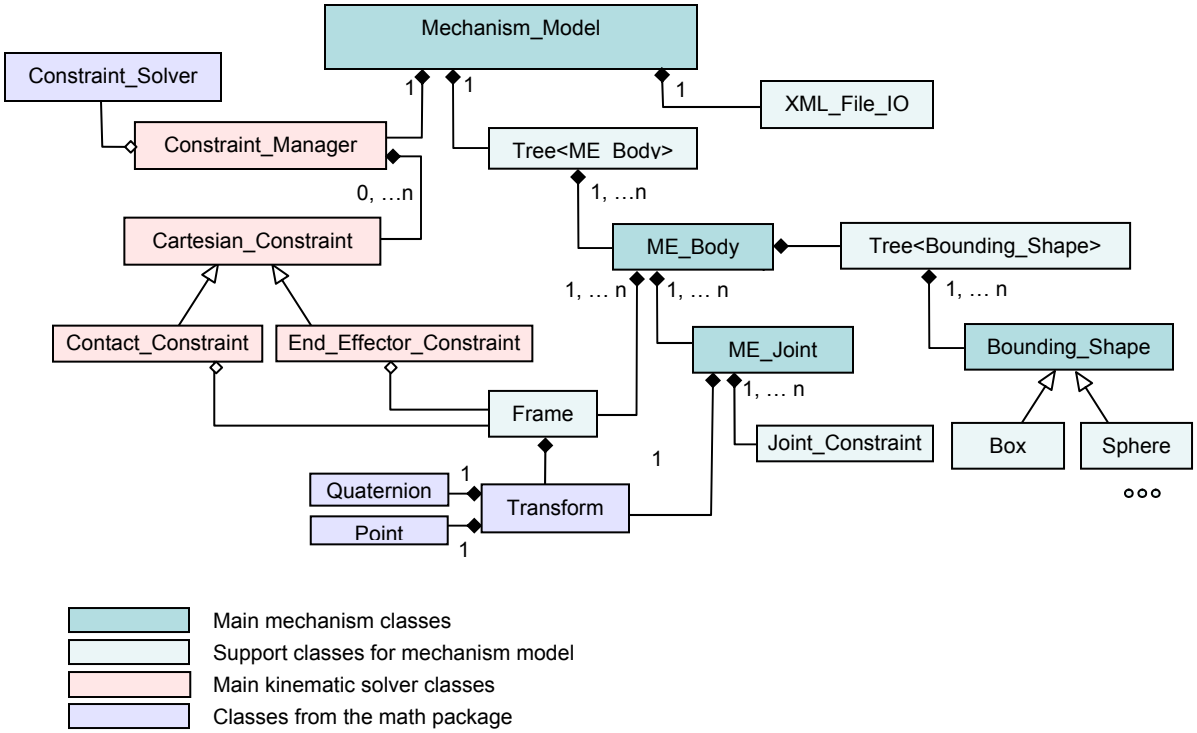
7.3. **Jacobian Matrix:** The `Mechanism_Model` object shall implement a function to return the differential relationship between articulation velocities and the linear and angular velocities of a Frame (i.e. the Jacobian matrix) for a given set of corresponding articulation values.

7.4. **Inverse Kinematics:** The `Mechanism_Model` object shall implement a function to compute the set of articulation values that will simultaneously fully or partially constrain one or more `Frame`(s) on the kinematic system to be located at other given `Frame` location(s)

7.5. The forward kinematics of 7.1 shall be also accessible through a public member function of `ME_Body`.

7.6. The differential forward kinematics of 7.2 shall be also accessible through a public member function of `ME_Body`

7.7. The Jacobian Matrix of 7.3 shall be also accessible through a public member function of `ME_Body`.



8. Constraint Management

8.1. A `Constraint_Manager` software object shall be used to administer the generic solution for inverse kinematics problems. The architecture will allow implementation of customized solutions in classes derived from `Mechanism_Model`.

8.2. In solving inverse kinematics problems, the `Constraint_Manager` shall use `Cartesian_Constraint` objects to specify desired relationships between pairs of frames.

8.3. Two types of `Cartesian_Constraint` objects shall be implemented:
- 8.3.1. Contact_Constraint objects
- 8.3.2. End_Effector_Constraint objects

8.4. `Contact_Constraint` objects shall be used to specify the desired surface contact between two frames.

8.5. `End_Effector_Constraint` objects shall be used to specify the desired absolute or relative position of a `Frame`.

8.6. For the solution of a general set of `Cartesian_Constraint` objects that simultaneously apply to the kinematic system, an iterative numerical approach shall be used.

8.7. The `Constraint_Manager` shall setup the `Cartesian_Constraint` vectors to solve for, and then use a `Constraint_Solver` to determine the configuration of the kinematic system that best solves for the set of `Cartesian_Constraint`s.

8.8. `Cartesian_Constraint` inputs may be entered from a file, a trajectory generator or serialized data input through a communication stream.

8.9. In the initial implementation, `Constraint_Solver` may use a simple numerical approach, for example a Newton-Raphson iteration, to solve for the set of constraints. However, the `Constraint_Solver` shall be extended to use other solvers for the constraints.

8.10. The implementation shall allow the user to easily replace this generic constraint solution approach with customized solutions for particular inverse kinematic problems.

9. Performance

These performance requirements are intended to be representative (but not exhaustive) of the acceptable overhead for the mechanism model classes. These timing requirements are based on a 1GHz x86 class processor with 128 MB of RAM

9.1. Retrieving kinematic and dynamic parameters from the mechanism model shall take less than 250 usec (TBD) per node for a 10 node tree topology.

9.2. Computing forward kinematics shall take less than 400 usec (TBD) for a 10 node serial chain

9.3. Computing generic inverse kinematics shall take less than 15 msec (TBD) for a 5 node serial chain.

10. Future Additions

- Add capability to specify collision matrix that lists pairs of objects that can potentially collide in an XML file

Part II: Interfacing Mechanism Model to CLARAty Abstractions

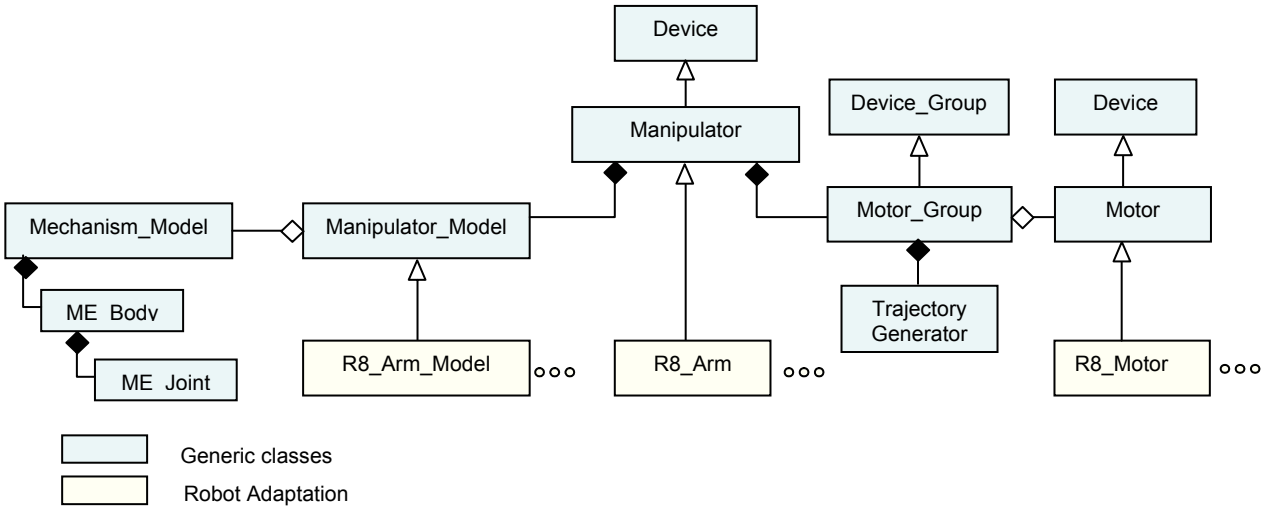


Figure 8: Example of using `Mechanism_Model` with `Manipulator` control classes. Similar structures will be used for Wheel_Locomotor and Legged_Locomotor classes (substitute manipulator with locomotor)Use of Mechan

The second part of this document describes how this mechanism model will interface with the CLARAty locomotion and manipulation abstracionts.

11. Interfacing with Control Abstractions

The class hierarchy for the `Mechanism_Model` software package is illustrated in . The `Mechanism_Model` classes shall be used either as a stand-alone package for kinematic analysis or as part of the control software for the robotic system.

11.1. `Mechanism_Model` may be used directly by the robot control system or through interface classes that present simpler and restricted interfaces for common types of mechanisms. For example, the interface class for manipulators shall be the `Manipulator_Model` class. Similarly, for wheeled mobile robots, it shall be `Wheeled_Locomotor_Model` and, for walking mobile robots, it shall be `Legged_Locomotor_Model`. Figure 8 shows an example using the manipulator classes to illustrate the interface of the mechanism model to the control classes (generic physical components).

11.2. The interface classes and their corresponding control classes shall enable the mechanical sub-systems to be controlled independently. This is the typical mode used for robotic operations. For example, a rover with its mounted arm can be treated as two independent control systems by utilizing the `Manipulator_Model` and `Wheeled_Locomotor_Model` interface classes for the arm and the rover respectively. Alternatively, one can use the `Mechanism_Model` instead to simultaneously coordinate the arm with the rover motions.

11.3. Adaptations of Manipulator_Model, Wheeled_Locomotor_Model and Legged_Locomotor_Model can override generic kinematics algorithm implementations with specialized algorithms whenever necessary.
11.4. Specialized kinematic algorithm implementations shall use parameter information from Mechanism_Model classes.
11.5. Manipulator_Model, Wheeled_Locomotor_Model and Legged_Locomotor_Model classes shall support serialization of their model information. This will support both writing these models to file storage and retrieving and instantiating the models from their respective files(see Section 6).

12. Model Classes for Manipulators and Locomotors

This section addresses models for generic manipulators, wheeled locomotors, and legged locomotors. We will use the Manipulator_Model class as an example. Figure 8 illustrates the relationships among Manipulator_Model interface class, Mechanism_Model class, and Manipulator control classes. The same applies for wheeled, legged and hybrid locomotors.

12.1. Manipulator_Model shall reference the Mechanism_Model class for accessing the kinematic model data on the manipulator.
12.2. Adaptations of the Manipulator_Model class may define specialized implementations (e.g., closed-form solution) of forward, inverse, and differential kinematics to override, as necessary, generic solutions available from the Mechanism_Model. Adaptations of the Wheeled_Locomotor_Model class shall define specialized implementations of flat-ground open-loop driving and slope differential driving kinematics algorithms. Adaptations of the Legged_Locomotor_Model class shall define specialized implementations of flat-ground open-loop walking and slope differential walking kinematics algorithms.
12.3. Manipulator_Model subclasses shall be concrete classes. Examples of such classes include R8_Arm_Model and FD_Mast_Model.

13. Control Classes for Manipulators and Locomotors

This section applies to the generic manipulator, wheeled locomotor, and legged locomotor classes. The requirements below are only intended to describe the relationship between the manipulator/locomotor classes and their corresponding model classes. The requirements below use the Manipulator class as an example. Figure 8 illustrates the Manipulator and the associated control classes. The same applies for wheeled, legged and hybrid locomotors.

13.1. Manipulator class shall inherit from generic Device class and contain the following:
- Manipulator_Model
- Motor_Group

13.2. Motor_Group class shall define the correspondence between manipulator joints and actual motors.
13.3. Manipulator class shall bind the ME_Joint objects to the individual motors
13.4. Manipulator class shall have APIs that support various motion control modes for:
13.4.1. Individual joint/wheel control
13.4.2. Coordinated joint/wheel control

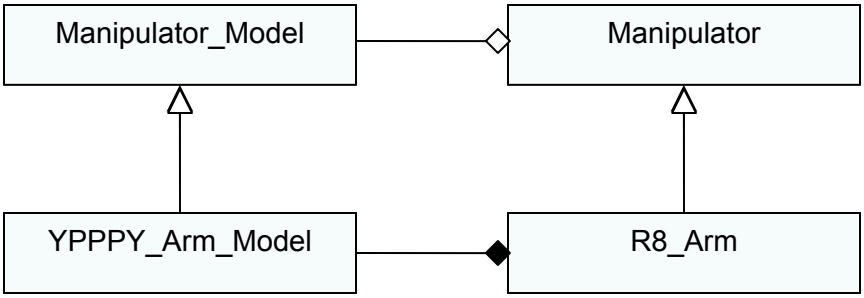


Figure 9: Inheriting specific models from `Manipulator_Model`

13.4.3. Following a prescribed path or trajectory (e.g. straight line motion of the end effector of a manipulator)
13.4.4. Gravity compensation for manipulators
13.4.5. Sensor-based control such as force control, compliance control, visual servoing, and so on.

13.5. `Motor_Group` may use a `Coordinator` object for closed loop coordination of multiple joints. The `Manipulator` class shall generate set points using generic or specialized kinematic algorithms and feed them to the `Coordinator` through the `Motor_Group` API.

13.6. `Manipulator` subclasses shall be concrete classes. Examples include `R8_Arm` and `FD_Mast`.

14. Examples of Model Instantiation

An example to create a mechanism model of a walking robot shall be as follows

```
// Create a mechanism model of the K-9 arm
YPPPY_Arm_Model ypppy_model("k9_arm_model.xml");

// Create a R7_Arm manipulator with a YPPPY model
R7_Arm          r7_arm(ypppy_model);
```

To create an arm that uses a yaw-pitch-pitch-pitch-yaw (YPPPY) configuration, one can create a YPPPY kinematic model which is a specialization of `Mechanism_Model`.

```
// Create a mechanism model of the rover body
Mechanism_Model   rover_model("body_model.xml");

// Attach a leg to the "leg1_mnt" frame defined in "body_model.xml"
// using the leg model in "leg_model.xml"
rover_model.attach(("leg_model.xml", "leg1_mnt");

// Attach a leg to the "leg2_mnt" frame defined in "body_model.xml"
// using the leg model in "leg_model.xml"
rover_model.attach(("leg_model.xml","leg2_mnt");
rover model.save("entire model.xml");
```

Appendix A: Document Definition Example

This appendix contains two examples of the mechanism model XML files, and their corresponding Document Type Definition for mechanism model. Below is the simple example:

```
<?xml version="1.0" encoding="US-ASCII" ?>
<!DOCTYPE Mechanism_Model SYSTEM "mechanism_model.dtd">
  <!--
   *=========================================================================
   *=                          /-/ CLARAty /-/                              =
   *=========================================================================
   * @file simple_sample1.xml
   *
   * An example of the use of the DTD
   *
   * @author:        Diaz-Calderon
   * @date:          June 15, 2004
   *
   * Software Use Notice
   * -------------------
   * http://claraty.jpl.nasa.gov/sw_use_notice.html or
   * ../share/sw_use_notice.txt
   *
   * (C) 2004, Jet Propulsion Laboratory, California Institute of Technology
   *
   * $Revision:$
   *-------------------------------------------------------------------------
  -->
<Mechanism_Model name="simple_two_link"  version="1.0">

<ME_Body name="link1">
   <ME_Joint name="joint1" type="revolute" actuated="true" offset="0"></ME_Joint>
   <Frame name="ref1" type="reference">
         <Transform>
               <Position x="2.0" y="0" z="0" />
               <Rotation type="Euler_ZYZ" angle="M_PI/2" angle="0" angle="0" />
         </Transform>
   </Frame>
</ME_Body>
<ME_Body name="link2" parent="link1">
   <ME_Joint name="joint2" type="revolute" actuated="true" offset="0"></ME_Joint>
   <Frame name="ref2" type="reference">
         <Transform>
               <Position x="10" y="0" z="0" />
               <Rotation type="Euler_XYZ" angle="M_PI/2" angle="M_PI" angle="0" />
         </Transform>
   </Frame>
</ME_Body>
</Mechanism_Model>
<!--
 EOF simple_sample1.xml
 -->
```

Below is a second more complete sample model for a three link manipulator.

```
<?xml version="1.0" encoding="US-ASCII" ?>
<!DOCTYPE Mechanism_Model (SYSTEM "mechanism_model.dtd">
  <!--
   *=============================================================================
   *=                          /-/ CLARAty /-/                                  =
   *=============================================================================
   * @file sample1.xml
   *
   * An example of the use of the DTD
   *
   * @author:        Diaz-Calderon
   * @date:          June 15, 2004
   *
   * Software Use Notice
   * -------------------
   * http://claraty.jpl.nasa.gov/sw_use_notice.html or
   * ../share/sw_use_notice.txt
   *
   * (C) 2004, Jet Propulsion Laboratory, California Institute of Technology
   *
   * $Revision:$
   *-----------------------------------------------------------------------------
  -->
<Mechanism_Model name="two_link" version="1.0">
<ME_Body name="link1">
  <ME_Joint name="joint1" home="1.0" type="revolute" actuated="true" offset="0">
      <Joint_Limits min="-M_PI" max="M_PI" vmax="0.25" torque_min="0"
      torque_max="0" />
      <Joint_Stiffness kx="0.1" ky="0.2" kz="0.3" />
  </ME_Joint>
  <Frame name="ref1" type="reference">
      <Transform>
          <Position x="0" y="0" z="0" />
          <Quaternion qi="0" qj="0" qk="0" qs="1" />
      </Transform>
  </Frame>
  <Mass_Properties mass="0.25">
      <CM>
          <Transform>
              <Position x="1" y="1" z="1" />
              <Quaternion qi="0" qj="0" qk="0" qs="1" />
          </Transform>
      </CM>
      <Inertia xx="0" yy="0" zz="0" />
  </Mass_Properties>
  <Bounding_Shape>
      <Obj_Params name="obj20" type="box" link="IDD_COL_EL_JOINT"
      is_container="true" level="1" is_aligned="no" ignore_collisions="false" tol="-
      1.0">
```

```
                <Transform>
                    <Position x="0" y="0" z="0" />
                    <Quaternion qi="0" qj="0" qk="0" qs="1" />
                </Transform>
                <Dimension dim-x="0" dim-y="0" dim-z="0" />
            </Obj_Params>
        - <Obj_Params name="obj21" type="box" parent="obj20"
            link="IDD_COL_LINK_AUTO" is_container="none" level="2" is_aligned="no"
            ignore_collisions="false" tol="-1.0">
                <Transform>
                    <Position x="0" y="0" z="0" />
                    <Quaternion qi="0" qj="0" qk="0" qs="1" />
                </Transform>
                <Dimension dim-x="0" dim-y="0" dim-z="0" />
            </Obj_Params>
-           <Obj_Params name="obj22" type="box" parent="obj20"
            link="IDD_COL_LINK_AUTO" is_container="none" level="1" is_aligned="no"
            ignore_collisions="false" tol="-1.0">
                <Transform>
                    <Position x="0" y="0" z="0" />
                    <Quaternion qi="0" qj="0" qk="0" qs="1" />
                </Transform>
                <Dimension dim-x="0" dim-y="0" dim-z="0" />
            </Obj_Params>
    </Bounding_Shape>
    <Display_Graphics path="graphics_file.vrml" />
  </ME_Body>
- <ME_Body name="link3">
-   <ME_Joint name="joint3" type="revolute" actuated="true" offset="0" home="0">
        <Joint_Limits min="-M_PI" max="M_PI" vmax="1.0" torque_min="0"
        torque_max="0" />
    </ME_Joint>
-   <Frame name="ref3" type="local">
        <Transform>
            <Position x="1" z="10" y="0" />
            <Quaternion qi="0" qj="0" qk="0" qs="1" />
        </Transform>
    </Frame>
-   <Mass_Properties>
-       <CM>
            <Transform>
                <Position x="0" y="0" z="0" />
                <Quaternion qi="0" qj="0" qk="0" qs="1" />
            </Transform>
        </CM>
        <Inertia xx="1" yy="2" zz="3" />
    </Mass_Properties>
  </ME_Body>
- <ME_Body name="link2" parent="link1">
-   <ME_Joint name="joint2" type="revolute" actuated="false" offset="M_PI_2">
```

```
        <Joint_Limits min="-M_PI" max="M_PI" vmax="1.0" torque_min="-1.0"
        torque_max="1.0" />
        <Joint_Stiffness kx="0.1" ky="0.2" kz="0.3" />
        <Joint_Constraint expr="2*joint1" />
    </ME_Joint>
-   <Frame name="ref2" type="reference">
        <Transform>
            <Position x="10" y="0" z="0" />
            <Quaternion qi="0" qj="0" qk="0" qs="1" />
        </Transform>
    </Frame>
-   <Frame name="sens1" type="local">
        <Transform>
            <Position x="4" y="10" z="0" />
            <Quaternion qi="0" qj="0" qk="0" qs="1" />
        </Transform>
    </Frame>
-   <Frame name="force1" type="local">
        <Transform>
            <Position x="3" y="2" z="0" />
            <Quaternion qi="0" qj="0" qk="0" qs="1" />
        </Transform>
    </Frame>
-   <Mass_Properties mass="0.25">
-       <CM>
            <Transform>
                <Position x="1" y="1" z="1" />
                <Quaternion qi="0" qj="0" qk="0" qs="1" />
            </Transform>
        </CM>
        <Inertia xx="0" yy="0" zz="0" />
    </Mass_Properties>
-   <Bounding_Shape>
-       <Obj_Params name="obj30" type="box" link="IDD_COL_EL_JOINT"
        is_container="true" level="1" is_aligned="no" tol="-1.0">
            <Transform>
                <Position x="0" y="0" z="0" />
                <Quaternion qi="0" qj="0" qk="0" qs="1" />
            </Transform>
            <Dimension dim-x="0" dim-y="0" dim-z="0" />
        </Obj_Params>
    </Bounding_Shape>
    <Display_Graphics path="graphics_file.vrml" />
</ME_Body>
</Mechanism_Model>
- <!--
 EOF sample1.xml
 -->
```

```
<?xml version="1.0" encoding="US-ASCII"?>

<!--
*=============================================================================
*=                          /-/ CLARAty /-/                                  =
*=============================================================================
* @file sample1.xml
*
* An example of the use of the DTD
*
* Target OS: Generic (VxWorks/UNIX/Linux)
*
* Designed by:    Diaz-Calderon
* @author:        Diaz-Calderon
* @date:          June 15, 2004
*
* Software Use Notice
* --------------------
* http://claraty.jpl.nasa.gov/sw_use_notice.html or
* ../share/sw_use_notice.txt
*
* (C) 2004, Jet Propulsion Laboratory, California Institute of Technology
*
* $Revision:$
*
* Modification History
* --------------------
* 002,18jun04,adc   Implemented changes per meeting with Issa
* 001,15jun04,adc   updated to conform with the requirements document
* 000,27apr04,adc   original writting of the dtd
*-----------------------------------------------------------------------------
-->

<!---------------- M E C H A N I S M _ M O D E L ----------------------->
<!ELEMENT Mechanism_Model ((ME_Body+) |
                           (Block_DH, Block_Joint))>
<!ATTLIST Mechanism_Model
  name    ID                                    #REQUIRED
  version CDATA                                 #REQUIRED>


<!---------------------- M E _ B O D Y ---------------------------------->
<!-- me-body requires at least one frame that describes the fixed -->
<!-- transform from parent reference frame to the body reference -->
<!-- frame. The required node will define the reference frame of the -->
<!-- body relative to the reference frame of the parent; i.e., the -->
<!-- body-to-parent transform. All other nodes (including mount nodes -->
<!-- must be defined relative to the body reference frame. -->
<!ELEMENT ME_Body (ME_Joint,
                   Frame+,
                   Mass_Properties?,
                   Bounding_Shape?,
                   Display_Graphics?)>
<!ATTLIST ME_Body
  name   ID                                     #REQUIRED
  parent IDREF                                  #IMPLIED>


<!---------------------- M E _ J O I N T -------------------------------->
<!ELEMENT ME_Joint (Joint_Limits?, Joint_Stiffness?, Joint_Constraint?)>
<!ATTLIST ME_Joint
  name       ID                                 #REQUIRED
  type       (revolute | prismatic) "revolute"
```

```
  actuated   (true | false) "true"
  offset     NMTOKEN "0"
  home       NMTOKEN "0">


<!-------------------- J O I N T _ L I M I T S -------------------->
<!ELEMENT Joint_Limits EMPTY>
<!ATTLIST Joint_Limits
  min        NMTOKEN "-M_PI"
  max        NMTOKEN "M_PI"
  vmax       NMTOKEN "1.0"
  torque_min NMTOKEN "0"
  torque_max NMTOKEN "0">


<!------------------- J O I N T _ S T I F F N E S S --------------->
<!ELEMENT Joint_Stiffness EMPTY>
<!ATTLIST Joint_Stiffness
  kx NMTOKEN                                  #REQUIRED
  ky NMTOKEN                                  #REQUIRED
  kz NMTOKEN                                  #REQUIRED>


<!------------------ J O I N T _ C O N S T R A I N T ------------->
<!ELEMENT Joint_Constraint EMPTY>
<!ATTLIST Joint_Constraint
  expr CDATA                                  #REQUIRED>


<!-------------------- C E N T E R  O F  M A S S ----------------->
<!-- coordinates of the center of mass relative to the body reference -->
<!-- frame. The orientation of the principal axes relative to the -->
<!-- reference frame is given by the rotation matrix in the -->
<!-- transform. -->
<!ELEMENT CM Transform>

<!------------------------ I N E R T I A ------------------------->
<!ELEMENT Inertia EMPTY>

<!-- Principal moments of inertia of the me-body. This assumes that -->
<!-- the frame located at the center of mass represent the principal -->
<!-- axes of the body. -->
<!ATTLIST Inertia
  xx NMTOKEN                                  #REQUIRED
  yy NMTOKEN                                  #REQUIRED
  zz NMTOKEN                                  #REQUIRED>

<!------------------ M A S S - P R O P E R T I E S --------------->
<!ELEMENT Mass_Properties (CM, Inertia)>

<!-- If the values for rho and volume are given, mass it is a derived -->
<!-- parameter mass = rho*volume -->
<!ATTLIST Mass_Properties
  rho    NMTOKEN                              #IMPLIED
  volume NMTOKEN                              #IMPLIED
  mass   NMTOKEN                              #IMPLIED
>


<!-------------------- B O U N D I N G _ S H A P E -------------->
<!ELEMENT Bounding_Shape (Obj_Params)+>

<!ELEMENT Obj_Params (Transform, Dimension)>
```

```
<!ATTLIST Obj_Params
  name              ID                          #REQUIRED
  parent            IDREF                       #IMPLIED
  type              (box | cyl | no_shape)      #REQUIRED
  is_container      (true | false | none)       #REQUIRED
  level             (1 | 2 | 3)                 #REQUIRED
  link              NMTOKEN                     #REQUIRED
  is_aligned        (yes | no)                  #REQUIRED
  ignore_collisions CDATA                       #REQUIRED
  tol               NMTOKEN                     #REQUIRED>


<!-------------- D I S P L A Y _ G R A P H I C S -------------------->
<!ELEMENT Display_Graphics EMPTY>
<!ATTLIST Display_Graphics
  path CDATA                                    #REQUIRED>



<!------------ P O S I T I O N S  &  V O L U M E S ---------------->

<!----------------------- P O S I T I O N ---------------------->
<!ELEMENT Position EMPTY>
<!ATTLIST Position
  x  NMTOKEN "0"
  y  NMTOKEN "0"
  z  NMTOKEN "0">

<!---------------------- D I M E N S I O N -------------------->
<!ELEMENT Dimension EMPTY>
<!ATTLIST Dimension
  dim-x  NMTOKEN "0"
  dim-y  NMTOKEN "0"
  dim-z  NMTOKEN "0">


<!-------------- R O T A T I O N S  &  F R A M E S ---------------->

<!------------------- X-A X I S  R O T A T I O N -------------------->
<!ELEMENT RX EMPTY>
<!ATTLIST RX
  angle NMTOKEN                                 #REQUIRED>

<!------------------- Y-A X I S  R O T A T I O N -------------------->
<!ELEMENT RY EMPTY>
<!ATTLIST RY
  angle NMTOKEN                                 #REQUIRED>

<!------------------- Z-A X I S  R O T A T I O N -------------------->
<!ELEMENT RZ EMPTY>
<!ATTLIST RZ
  angle NMTOKEN                                 #REQUIRED>

<!----------------------- 3D R O T A T I O N  ----------------------->
<!ELEMENT Rotation EMPTY>
<!ATTLIST Rotation
  type ( EULER_XYZ | EULER_ZYX | EULER_ZYZ )
  angle NMTOKEN                                 #REQUIRED
  angle NMTOKEN                                 #REQUIRED
  angle NMTOKEN                                 #REQUIRED>
```

```
<!---------------------- Q U A T E R N I O N ------------------->
<!ELEMENT Quaternion EMPTY>
<!ATTLIST Quaternion
  qi NMTOKEN "0"
  qj NMTOKEN "0"
  qk NMTOKEN "0"
  qs NMTOKEN "1">

<!----------------------   T R A N S F O R M -------------------->
<!-- Transform that uses a position and either a quaternion
<!-- or a rotation matrix for its orientation -->
<!ELEMENT Transform (Position, ( Rotation | Quaternion )) >


<!---------------------- D H _ P A R A M E T E R ----------------->
<!-- Denahvit-Hartenberg parameters which can use either Craig's or  -->
<!-- Paul's convention -->
<!ELEMENT DH_Parameter EMPTY>
<!ATTLIST DH_Parameter
  type   ( DH_Craig | DH_Paul )               #REQUIRED
  name   ID                                   #REQUIRED
  length NMTOKEN                              #REQUIRED
  twist  NMTOKEN                              #REQUIRED
  offset NMTOKEN "0"
  angle  NMTOKEN "0">

<!--------------------------- F R A M E ---------------------->
<!-- a Frame represents a ref-frame, actuator-frame and -->
<!-- sensor-frame. There can only be one and only one ref-frame for a -->
<!-- given body. It is an error to define more than one reference -->
<!-- frames in a body. -->
<!ELEMENT Frame ( Transform | DH_Parameter)>
<!ATTLIST Frame
  name    ID                                  #REQUIRED
  type    (reference | local) "local">


<!--------------------- T A B L E  I N P U T ------------------->

<!----------------------- B L O C K - D H --------------------->
<!ELEMENT Block_DH (DH_Parameter+)>

<!---------------------- B L O C K - J O I N T ----------------->
<!ELEMENT Block_Joint (ME_Joint+)>


<!-- EOF mechanism_model.dtd -->
```

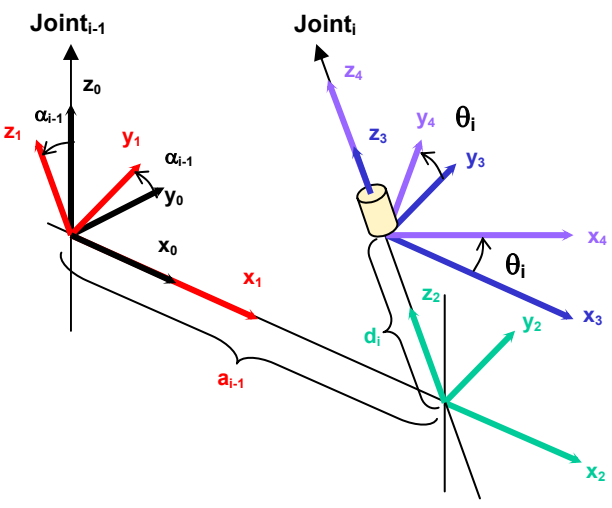
Appendix B: Model Input Representations

This appendix summarizes three input respresentations: (a) Craig's D-H parameters, (b) Paul's D-H parameters, and (c) Zero Position parameters.

Craig's definition of D-H parameters

T1 = twist α_{i-1} about x_0 axis
T2 = length a_{i-1} along x_1 axis
T3 = offset d_i along z_2 axis
T4 = rotate θ_i about z_3 axis

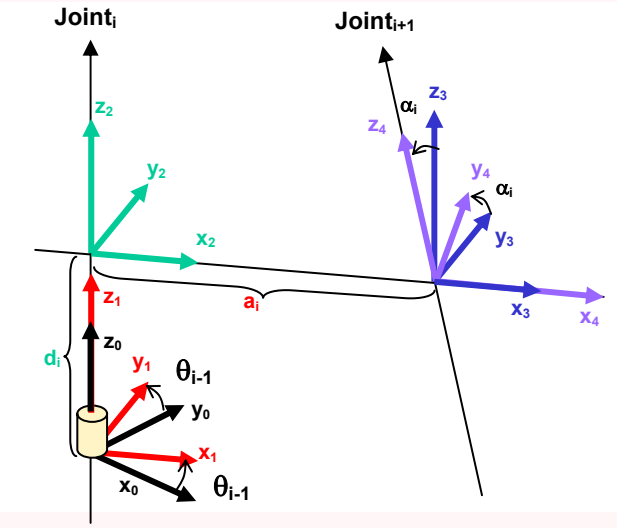
$$T_i^{i-1} = \begin{bmatrix} c\theta_i & -s\theta_i & 0 & a_{i-1} \\ s\theta_i\, c\alpha_{i-1} & c\theta_i\, c\alpha_{i-1} & -s\alpha_{i-1} & -d_i\, s\alpha_{i-1} \\ s\theta_i\, s\alpha_{i-1} & c\theta_i\, s\alpha_{i-1} & c\alpha_i & d_i\, c\alpha_{i-1} \\ 0 & 0 & 0 & 1 \end{bmatrix}$$



Paul's definition of D-H parameters

T1 = rotate θ_i about z_0 axis
T2 = offset d_i along z_1 axis
T3 = length a_i along x_2 axis
T4 = twist α_i about x_3 axis

$$T_{i+1}^{i} = \begin{bmatrix} c\theta_i & -s\theta_i\, c\alpha_i & s\theta_i\, s\alpha_i & a_i\, c\theta_i \\ s\theta_i & s\theta_i\, c\alpha_i & -s\theta_i\, s\alpha_i & a_i\, s\theta_i \\ 0 & s\alpha_i & c\alpha_i & d_i \\ 0 & 0 & 0 & 1 \end{bmatrix}$$



Zero Position Parameters (Darts Style)

In the initial (zero) position, the orientations of all coordinates frames are aligned

Body name = Link$_i$
Parent body name = Link$_{i-1}$
Joint type = rotational

Body to joint = xb, yb, zb (default = 0 0 0)
Parent body to joint = xp, yp, zp (default = 0 0 0)
Joint axis = ax, ay, az